Applying the Decorator Pattern for Profiling Object-Oriented Software

Edward B. Duffy
Computer Science Dept.
Clemson University
Clemson, SC 29634
USA
eduffy@cs.clemson.edu

J. Paul Gibson
Computer Science Dept.
National University of Ireland
Maynooth, Co. Kildare
Ireland
Paul.Gibson@may.ie

Brian A. Malloy
Computer Science Dept.
Clemson University
Clemson, SC 29634
USA
malloy@cs.clemson.edu

Abstract

A profiler can provide valuable information to a developer to facilitate program optimization, debugging or testing. In this paper, we describe the use of the Decorator pattern for non-intrusive profiling of object-oriented applications. We provide a formal specification of the Decorator pattern, and show that the pattern can be used as a program transformation without altering the external, observable behavior of the system. We refer to such a transformation as a correctness preserving transformation, *or CPT. As a CPT, the Decorator pattern can be used to non-intrusively profile object-oriented applications and we illustrate this application with an invariant validator for enforcement of Design by Contract, and for profiling memory. We provide a case study to compare the cost trade-offs of validating invariants at different points in a program.*

1 Introduction

A program profile is a description of the relative frequency of program regions or the frequency of resource usage. Profilers take many forms and each form can be a valuable tool that can provide useful guidance in program optimization [9, 23], debugging [3, 19, 28] and testing [20].

There are two approaches for implementing profilers: the probabilistic approach and the instrumentation approach [19]. In the *probabilistic approach*, a system-level interrupt handler is periodically invoked and run-time state information is recorded. The record of each interrupt is termed a *sample* and the samples for the entire program execution form the profile. In the *instrumentation approach*, counters are inserted into the text of the program and these frequency counts form the profile of the program. The disadvantage of these approaches is that each technique perturbs the program under profile and this perturbation can turn a correct program into a faulty one [19].

In this paper, we describe the use of the Decorator pattern for non-intrusive profiling of object-oriented applications. We provide a formal specification of the Decorator pattern, and show that the pattern can be used as a program transformation without altering the external, observable behavior of the system. We refer to such a transformation as a *correctness preserving transformation*, or CPT [16]. As a CPT, the Decorator pattern can be used to non-intrusively profile object-oriented applications and we illustrate this application with an invariant validator for enforcement of Design by Contract [26], and for profiling memory. We provide a case study to compare the cost trade-offs of validating invariants, expressed in the object constraint language (OCL) [7], at different points in a program.

In the next section we provide background and review the work related to profiling programs. Section 3 provides an overview of the Decorator pattern and describes its use in profiling. We formalize the Decorator pattern as a correctness preserving transformation in reference [10]. In Sections 4 and 5 we describe the use of the Decorator pattern for validating invariants and profiling memory and in Section

1092-8138/03 $17.00 © 2003 **IEEE**
IEEE COMPUTER SOCIETY

6 we present the results of our case study. Finally, in Section 8, we draw conclusions and describe our ongoing work.

2 Background and Related Work

In this section we provide background and overview the research that relates to program profiling and validating invariants. Since profiling memory is a minor part of our paper, we omit background and related work about this topic.

2.1 Profiling programs

The advantages of profiling programs have long been recognized for their abilities to facilitate optimization, debugging and testing [23]. Fine-grained profile descriptions are obtained by instrumenting the program to measure execution counts of basic blocks [5], control-flow edges [5] or paths [6] in a program. Function profilers provide a more coarse-grained description of the program by measuring, for each function, the total time spent executing the function, possibly including the time spent in its call descendants on its behalf [1, 31, 4].

More recent research has focused on providing more comprehensive profile information. Reference [19] describes the computation of *call path refinement profiles*, which provide detailed performance information about arbitrarily nested function call sequences, with the ability to unravel the sequences either backward or forward in the call stack, allowing the programming to detect the bottlenecks in the code.

Reference [28] exploits profile information to compare two different executions of a single program with input data that differs only on the date. Reference [20] compares executions of a program and a modified version of the program on the same input data in an effort to uncover regression faults. Both of these approaches use program profiling to obtain a distribution of the profile information called a *spectrum*; different spectra are compared to obtain information about the program.

The approach to program profiling that is most similar to ours is described in reference [11], where the Decorator pattern is used to implement violation checking wrappers for Java components. Our use of the Decorator pattern is similar, except that we generalize the use of the Decorator to profiling of class methods. Moreover, we formalize our use of the Decorator for profiling as a correctness preserving transformation that maintains the observable behavior of the original program.

2.2 Invariant validation

An invariant on a class C is a set of Boolean conditions or predicates that every instance of C will satisfy after instantiation (i.e., after constructor invocation) and before and after every method invocation by another object [26]. A class invariant is a property of a class instance that must be preserved by all methods of the class. In spite of its name, an invariant is not required to hold at all execution points. For example, a method might violate the invariant while working toward its goal; however, the invariant must be re-established before the method terminates execution.

Reference [14] describes a non-invasive approach for validation of class invariants in C++ applications, where the invariants are specified using the Object Constraint Language (OCL). The approach is fully automated so that the user need only supply the class invariants for each class hierarchy to be checked and a validator constructs an *InvariantVisitor*, a variation of the Visitor pattern [25], and an *InvariantFacilitator*. Instantiations of the *InvariantVisitor* and *InvariantFacilitator* classes encapsulate the invariants in C++ statements and facilitate the validation of the invariants. However, the technique in reference [14] does not easily extend for validating invariants at arbitrary points in a program.

3 Overview of the Technique

In this section, we overview the Decorator pattern [12] and describe its use in profiling object-oriented applications. The *Decorator pattern* is ideal for attaching responsibilities to objects transparently. We apply the Decorator to profiling, enabling us to track objects without affecting the object. Moreover, we can withdraw the profiling, with minimal modification to the application.

3.1 Structure of the pattern

The class diagram in Figure 1 describes the structure of the Decorator pattern [12] that we use for

Proceedings of the 11 th IEEE International Workshop on Program Comprehension (IWPC'03)
1092-8138/03 $17.00 © 2003 IEEE
IEEE COMPUTER SOCIETY

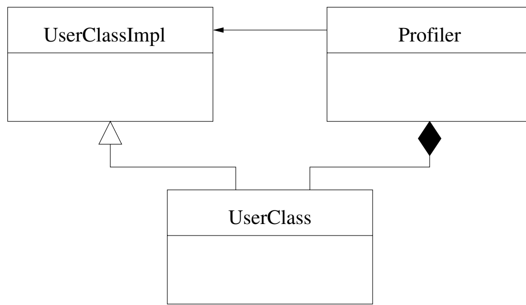

Figure 1. *Pattern structure*. **This figure illustrates the structure of the Decorator pattern, as applied to program profiling.** `UserClassImpl` **is the original user-written class under profile,** `Profiler` **implements the profiler, and** `UserClass` **unifies classes** `UserClassImpl` **and** `Profiler`**.**

profiling. There are three classes in the figure: `UserClassImpl`, representing the original user-written class under profile; `Profiler`, the class containing the code to perform profiling; and `UserClass`, a class that unifies the class under profile and the profiler. Class `UserClass` inherits publicly from `UserClassImpl` and contains `Profiler`; these relationships enable the pattern to be used by object-oriented languages that might not include multiple inheritance.

To describe our approach to profiling, assume the user intends to profile a class `UserClass`. We begin by renaming `UserClass` to `UserClassImpl`, and subclass `UserClassImpl` with a new class called `UserClass`, which contains an instance of the class `Profiler`. In the new `UserClass`, we overload the functions to be profiled from the original class; in the overloaded functions, we invoke the profiler as well as the original member function. In the next section, we provide an example of this approach

3.2 An example: profiling a traffic light

Figure 2 illustrates C++ code containing a namespace, lines 1 through 5, a class to be profiled, lines 6 through 19, and a main program, lines 20 through 25. The namespace, `LightColors`, contains an enumeration, `colors`, with a value for the three colors of a traffic light: `GREEN`, `YELLOW` and `RED`, as well as an array, `colorNames_`, to facilitate printing of the

```
 1 namespace LightColors {
 2    enum colors { GREEN, YELLOW, RED, NUMBER_OF_COLORS };
 3    const string colorNames_[] =
 4            { "Green", "Yellow", "Red" };
 5 };

 6 class TrafficLight {
 7 public:
 8    TrafficLight(LightColors::colors color
 9               = LightColors::GREEN)
10       : color_(color) { }
11    void Switch() {
12       color_ = static_cast<LightColors::colors>(
13          (color_ + 1) % LightColors::NUMBER_OF_COLORS);
14       cout << LightColors::colorNames_[color_]
15            << endl;
16    }
17 private:
18    LightColors::colors color_;
19 };

20 int main() {
21    TrafficLight light(LightColors::RED);
22    for (int i(0); i < 10; ++i) {
23       light.Switch();
24    }
25 }
```

Figure 2. *A Traffic Light example*. **This figure shows a traffic light that switches colors. The traffic light example threads our paper.**

enumeration.

The class in Figure 2, `TrafficLight`, contains a data member, `color_`, which takes on the values of the `colors` enumeration. `TrafficLight` also contains two member functions, a constructor that initializes `color_` and `Switch()`, which changes the value (color) of the simulated traffic light and prints the current color. Finally, function `main`, lines 20 through 25 of Figure 2, instantiates the `TrafficLight`, `light`, and changes the light ten times.

Figure 3 illustrates our profiling of the traffic light in Figure 2. Class `TrafficLightImpl` on lines 5 through 17, is the same as class `TrafficLight` in Figure 2, except that it has been renamed. `TrafficLightImpl` contains member function `Switch()`, as in Figure 2; in this example, we profile the actions of function `Switch()`.

Class `Counter`, lines 18 through 28, models our `Profiler` class of Figure 1; we use `Counter` in the traffic light example to monitor the traffic light. Class `Counter` contains an integer data member, `count_`, which monitors changes to the light, and a file, `log_`,

1092-8138/03 $17.00 © 2003 IEEE

```
1  namespace LightColors {
2     enum colors { GREEN, YELLOW, RED, NUMBER_OF_COLORS };
3     const string colorNames_[] = { "Green", "Yellow", "Red" };
4  };

5  class TrafficLightImpl {
6  public:
7     TrafficLightImpl(LightColors::colors color = LightColors::GREEN)
8        : color_(color) {}
9     void Switch() {
10       color_ = static_cast<LightColors::colors>( (color_ + 1) %
11          LightColors::NUMBER_OF_COLORS);
12       cout << LightColors::colorNames_[color_]
13            << endl;
14    }
15 private:
16    LightColors::colors color_;
17 };

18 class Counter {
19 public:
20    Counter() : count_(0), log_("light.log") { }
21    void Increment() {
22       ++count_;
23       log_ << count_ << endl;
24    }
25 private:
26    int count_;
27    ofstream log_;
28 };

29 class TrafficLight : public TrafficLightImpl,
30                      private Counter {
31 public:
32    TrafficLight(LightColors::colors color)
33                : TrafficLightImpl(color) {}
34    void Switch() {
35       Increment();
36       TrafficLightImpl::Switch();
37    }
38 };

39 int main() {
40    TrafficLight light(LightColors::RED);
41    for (int i(0); i < 10; ++i) {
42       light.Switch();
43    }
44 }
```

Figure 3. *A Profiled Traffic Light*. **This figure illustrates our application of the Decorator pattern to profiling the traffic light of Figure 2. Class** `TrafficLightImpl` **on lines 5 through 17, is the same as class** `TrafficLight` **in Figure 2, except that it has been renamed.** `TrafficLightImpl` **contains member function** `Switch()`**, as in Figure 2; in this example, we profile the actions of function** `Switch()`**. Class** `Counter`**, lines 18 through 28, models our** `Profiler` **class of Figure 1; we use** `Counter` **in the traffic light example to monitor the traffic light. Function** `Switch()`**, lines 34 to 37 in** `TrafficLight`**, first profiles the traffic light by incrementing** `count_` **in** `Counter`**, and then invokes** `TrafficLightImpl::Switch()`**, to implement the functionality of the original** `Switch()`**.**

Proceedings of the 11 th IEEE International Workshop on Program Comprehension (IWPC'03)
1092-8138/03 $17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

which stores logged information. `Counter` contains a constructor that initializes the data members and function `Increment()` that might be used to monitor changes to a class data attribute.

Class `TrafficLight`, lines 29 through 38 of Figure 3, is a new class that unifies the class under profile, now called `TrafficLightImpl`, and the profiler, that we call `Counter` in this example. Class `TrafficLight` uses multiple inheritance to unify the two classes, line 29 of Figure 3, where `TrafficLight` inherits publicly from `TrafficLightImpl` and privately from `Counter`. Our use of multiple inheritance in this figure is equivalent to the inheritance and aggregation illustrated in the class diagram of Figure 1.

Class `TrafficLight` has no data attributes and two member functions, a constructor that simply passes the initial value of the traffic light to its parent class, and `Switch()`, which overloads the function in the parent class. Function `Switch()` in `TrafficLight` first profiles the traffic light by incrementing `count_` in `Counter`, and then invokes `TrafficLightImpl::Switch()`, to implement the functionality of the original `Switch()`. Thus, in this example we first profile the traffic light and then execute the original functionality.

4 Using the Decorator Pattern to Validate Invariants

Class invariants are usually defined as constraints or restrictions on the state of the objects in the class. They are most useful when the state of an object is composed of component objects. The object is considered to be broken, or unsafe, if the component states do not meet the invariant property. For example, we may define a road junction as a composition of two traffic lights. The junction would probably be unsafe if both lights were green at the same time, and in such an instance the developers should specify this requirement as an invariant property.

Invariant properties that are not formally verified introduce the possibility of *run time* errors in an execution model. We promote a complementary approach to checking invariants. The engineers should try to verify them at compile time (using formal methods), but where this is not feasible (or practical) then the invariants that have not been verified should be tested through some run-time validation mechanism. There are a number of alternatives for how this can be done (which we discuss in the case study of Section 6). For now we show that the Decorator pattern can be used for such run-time validation. For the purpose of this example, we chose to check invariants as pre-conditions to method calls and log any instances where invariants are broken. This profile log does not guarantee the absence of errors due to broken invariants but it will either increase our confidence in the code being correct, or quickly identify a broken invariant for immediate repair.

We now address the validation and verification of such invariant properties in OO ACTONE [15]. There are four parts to this problem. First, we must consider the *property language* for specifying invariants. Second[1], we must consider how to use a theorem prover for the *automated verification* of these properties. Third, we must consider how to validate those invariants that we have been unable to verify (through proof). Finally, we must consider how to implement this validation mechanism.

4.1 Run-time validation with profiler: the specification

In this section we specify an invariant for a traffic light junction, illustrated in Figure 4, and show how it can be formulated for validation using the Decorator pattern. Reference [10] contains a more thorough discussion of this formulation as a correctness preserving transformation.

Note that we specify an initial state and that this satisfies the invariant property. It is easy, in this example, to see that the invariant property is going to be broken after the first call to Switch, lines 9 and 10, at the junction: both lights turn green. The specification requires that this be logged by the profiler. (In an implementation we would probably log more information in order for us to identify the error.)

In this simple example, it is more than likely that the error would be found immediately by a theorem prover. However, for the sake of the example we imagine that a tool for formal verification is not available. Thus we are required to validate the invariant at run-time. The Decorator pattern facilitates immediate implementation. A Java implementation of the Decorator pattern, instantiated

[1]It is beyond the scope of this work to examine this process in detail: see reference [17].

Proceedings of the 11 th IEEE International Workshop on Program Comprehension (IWPC'03)
1092-8138/03 $17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

```
1  CLASS Junction USING TrafficLight
2  STRUCTURES A_Junction(TrafficLight, TrafficLight)
3     INITIALIZE A_Junction(A_TrafficLight(Red),
4        A_TrafficLight(Red))
5     INVARIANT A_Junction(tl1,tl2) REQUIRES
6        tl1=Red or tl2 =Red
7     TRANSFORMERS Switch()
8  EQUATIONS
9     A_Junction(tl1, tl2).Switch()=
10       A_Junction(tl1.Switch(), tl2.Switch())
11 ENDCLASS Junction

12 CLASS Profiler USING Stream
13 STRUCTURES A_Profiler(Stream)
14    TRANSFORMERS Note()
15    EQUATIONS
16       A_Profiler(log).Note() =
17             A_Profiler(log.add("Invariant Broken"));
18 ENDCLASS Profiler

19 CLASS ProfiledJunction EXTENDS Junction, Profiler
20 TRANSFORMERS Note HIDDEN
21 STRUCTURES A_ProfiledJunction(Junction, Profiler)
22    EQUATIONS
23    j.INVARIANT() => A_ProfiledJunction(j,p).Switch()
24       = A_ProfiledJunction(j.Switch(),p);
25    not(j.INVARIANT()) =>
26       A_ProfiledJunction(j,p).Switch()
27          = A_ProfiledJunction(j.Switch(),p.Note());
28 ENDCLASS ProfiledTrafficLight
```

Figure 4. A Traffic Light example. This figure shows a traffic light that switches colors. This example is used throughout our paper.

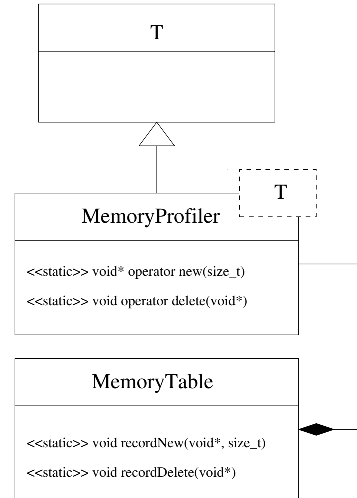

Figure 5. *Profiling Memory*. **This figure illustrates an application of the Decorator pattern for profiling memory in C++ applications.**

Test case	lines	classes	classes w/ fns
encrypt	946	1	1
php2cpp	1,920	6	6
fft	2,238	51	36
graphdraw	4,354	199	76
ep matrix	4,944	78	51
vkey	8,556	279	44

Figure 6. *Test suite*.

by the Junction and the Profiler as specified above, is presented in reference [10].

5 Using the Decorator Pattern to Profile Memory

We now exploit the Decorator pattern to profile memory in C++ applications. To accomplish this, we track memory allocation and deallocation by overloading operators *new* and *delete* for the class under profile. Unlike typical profiling, in memory profiling we know the methods that we wish to monitor, since they are provided by all classes, either as user-defined or compiler generated operators. We construct a generic class to profile the memory of the parameter to the generic class by monitoring operators *new* and *delete* of the generic class.

Figure 5 summarizes an approach to profiling memory, where we illustrate overloading for the simplest form of *new* and *delete*; the other three forms follow in similar fashion. The base class in the figure, `T`, is the user class under profile and the generic derived class, `MemoryProfiler`, is the profiled user class. Classes `T` and `MemoryProfiler` correspond to `UserClassImpl` and `UserClass` illustrated in Figure 1 of Section 3. The third class in Figure 5 is `MemoryTable`, which tracks memory, and corresponds to class `Profiler` in Figure 1.

6 Case Study: Validating Invariants

In this section we describe the results of our study of automated validation of class invariants, expressed

Proceedings of the 11 th IEEE International Workshop on Program Comprehension (IWPC'03)
1092-8138/03 $17.00 © 2003 IEEE
IEEE COMPUTER SOCIETY

in the object constraint language (OCL) [7]. The target application for our study is *keystone* [24, 27], a parser front-end for the ISO C++ language [21]. The validator was executed on a Sager NP5620 Laptop with a P4, 2.0 GHz processor, equipped with 256 MB of DDR, running the Linux operating system, kernel version 2.4.19. Our implementation language was C++ [30] compiled with GNU *gcc* version 3.2. In the next section we describe the test suite for the study and in Section 6.2 we compare the cost of validating invariants at various points in the program. In Section 6.3 we discuss the errors that were uncovered in the application for our study, and in Section 6.4 we discuss the code coverage provided by our test suite and the implication of coverage on invariant validation. In Section 6.5 we discuss threats to the validity of our study.

6.1 The test suite

The table in Figure 6 summarizes our suite of six test cases, listed in the rows of the table as encrypt, php2cpp, fft, graphdraw, ep matrix and vkey. The test cases in the suite were chosen because of their range and variety of application; they are listed in sorted order by number of lines of code, not including comments or blank lines. We note that keystone had been previously tested using this same test suite and was thought to run successfully.

Test case encrypt is an encryption program that uses the Vignere algorithm [2] and the php2cpp test case converts the PHP web publishing language to C++ [8]. fft performs fast Fourier transforms [22] and graphdraw is a drawing application that uses *IV Tools* [32], a suite of free XWindows drawing editors for Postscript, TeX and web graphics production. The ep matrix test case is an extended precision matrix application that uses *NTL*, a high performance portable C++ number theory library [29]. vkey is a GUI application that uses the *V GUI* library [33], a multi-platform C++ graphical interface framework to facilitate construction of GUI applications.

The columns of the table in Figure 6 list details about the number of lines of code, not including comments or blank lines, the number of classes, and the number of classes with functions for each of the test cases. All of the test cases are complete applications and three use large libraries: ep matrix, vkey and graphdraw use the *NTL*, *V GUI* and *IV Tools* libraries respectively.

Test case	ATT	EOM	EOP	NONE
encrypt	73.30	6.76	1.27	1.27
php2cpp	79.13	7.74	2.18	1.92
fft	120.30	13.33	3.72	3.22
graphdraw	875.40	33.26	6.69	6.66
ep matrix	863.50	98.07	47.58	46.6
vkey	316.70	44.44	21.54	21.48

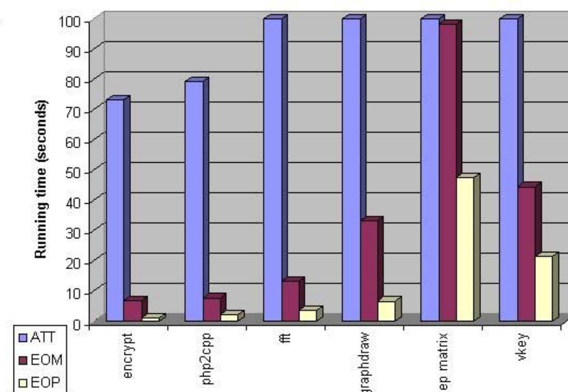

Figure 7. *Efficiency*.

6.2 Efficiency

In this section, we evaluate the performance of invariant validation at different points in the program. In particular, we use the Decorator pattern to profile invariant validation: (1) at the end of constructor execution, at the beginning of destructor execution and at the beginning and at the end of all methods, which we refer to as "All The Time", or ATT; (2) at the end of mutator methods, which we refer to as "End Of Mutator", or EOM; finally, (3) we compare our results for EOM and ATT with the approach of reference [14], where invariants are validated only at the end of the program, EOP.

Figure 7 contains a table, listing execution times for the various approaches, as well as a graph summarizing the results. There are five columns in the table. The first column lists the test cases, the second column lists results for ATT, the third column lists results for EOM, the fourth column lists results for EOP, and the final column lists execution times for the test case with no invariant validation, shown as NONE. For example, consider the first row of the table, showing results for the encrypt test case, where 73.30 seconds were required to validate invariants ATT, 6.76 seconds were required to validate invariants EOM, 1.27 seconds were required to

Proceedings of the 11 th IEEE International Workshop on Program Comprehension (IWPC'03)
1092-8138/03 $17.00 © 2003 IEEE
IEEE COMPUTER SOCIETY

validate invariants at EOP, and 1.27 seconds were required to parse encrypt without invariant validation.

The graph in Figure 7 summarizes our results for each test case. There are three sets of bars for each test case. The first bar of each set summarizes results for ATT, the second bar for EOM and the third bar for EOP. The first two sets of bars are drawn to scale, but the final four sets of bars abbreviate the results for ATT since, drawn to scale, the ATT bar would dwarf the other two bars.

The results in Figure 7 show that validating invariants ATT generally requires an order of magnitude longer than validating at EOM. Moreover, the time to validate invariants at EOM is usually more than double the time to validate only at EOP.

6.3 Impact of validation on keystone

We have chosen, for our case study, *keystone*, a parser front-end for the ISO C++ language [24, 27]. As an application, *keystone* instantiates objects representing name declarations, type declarations or scopes for the program under parse. These objects, once instantiated, remain viable until program termination; lookup is the typical operation on these objects, with updates performed less frequently. However, all of these objects remain until program termination.

Given the nature of our case study, we partition the errors exposed by invariant validation into three categories: (1) temporary errors, (2) mismatch errors, and, (3) implementation errors. *Temporary errors* result from names declared in the program under parse, but not yet defined. For example, some scope objects, such as a forward class declaration, are declared but defined later; thus, none of the values for the data attributes are available when the class object is constructed. Therefore, fields that are specified by an invariant to never be NULL, must be set to NULL temporarily, until the actual class declaration is encountered in the program.

Another example of a *temporary* error results from a name declaration that is instantiated to facilitate name lookup, but the declaration of the name has not yet been encountered; for example, a class data attribute may be initialized for a class before the declaration of the attribute is encountered. This type of error has prompted us to investigate the notion of *temporal invariants*, or invariants that are *eventually* valid [13]. We uncovered many temporary errors during invariant validation of *keystone*; all of these errors eventually, as the test case continues to be parsed, either fall into the second or third category of errors, or become valid.

The second category of errors, uncovered by invariant validation, are *mismatch errors*, which expose a discrepancy between the specification and the implementation. We uncovered 5 mismatch errors, the same number reported in reference [14]. The third category, *implementation errors*, represent errors in the code that were not exposed by testing.

6.4 Code coverage

Our approach to invariant validation is a dynamic activity and is therefore dependent on the coverage provided by our suite of test cases. To illustrate the importance of coverage, consider that the ISO C++ standard specifies that only labels have function scope. Thus, if none of the test cases in the testsuite have labels, then invariants defined on function scope will not be validated. Using the *gcov* utility, provided by *gcc*, the test suite listed in Figure 6 covered *92.2%* of the statements in the *keystone* application. Moreover, *100%* of the invariants were exercised one or more times during execution of the testsuite.

6.5 Threats to validity

There are threats to the validity of our case study, posed by the monotonic nature of the *keystone* application. In particular, the objects that we validate in *keystone*, once instantiated, are not destroyed until program termination. We have shown in Section 6.3 that we were unable to uncover additional errors by exercising the invariants more often than simply at program termination. However, in an application where objects are frequently created and destroyed during program execution, it is possible that only a small percentage of the objects will remain at program termination. For such an application, invariants might require more frequent validation, for example at termination of mutator functions or, perhaps, *all of the time*.

Moreover, the objects in the *keystone* application may have temporary state while in flux, but once the data attributes are set, remain stable. With objects whose state fluctuates, it is possible that invariants

Proceedings of the 11 th IEEE International Workshop on Program Comprehension (IWPC'03)
1092-8138/03 $17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

may be, at first, valid, but may then become invalid even across method invocation, and then become valid again. The invalid state will not be recognized if invariants are only validated upon program termination.

7 Implementation Difficulty

In this section, we overview two difficulties that we encountered during the implementation of the profiler. While our implementation language is C++ [30], the problems also occur in Java [18]. The first difficulty concerns run-time type information and the second difficulty concerns name hiding.

For the first difficulty, consider Figure 1, containing classes `UserClassImpl`, `Profiler`, and `UserClass`, described in Section 3.1. Our technique is designed so that a programmer may use either `UserClassImpl` or `UserClass` transparently. For most classes, this can be achieved by having `UserClass` simply forward function calls up to it's parent class (`UserClassImpl`). However, consider a class hierarchy that permits a polymorphic copy via a `Clone` method.

```
1  class XBase {
2  public:
3    virtual XBase * Clone() const = 0;
4  };
5  class XImpl : public XBase {
6  public:
7     XBase * Clone() const {
8        return new XImpl(*this);
9     }
10 };
```

A naive implementation of `X` would simply call `XImpl::Clone()` and return its value. However, a run-time type query on the object will return incorrect information, such as `dynamic_cast` in C++ and the `instanceof` operator in Java. For example, the type of the object returned by `XImpl::Clone` is an `XImpl`, but the user of the class should not have any knowledge of `XImpl`, rather just `X` and would expect that to be the type of the object. We could rewrite the body of `XImpl::Clone` in the declaration of `X` to make sure the object type is correct, but keeping duplicate copies of code may introduce a maintenance problem. Thus, we describe a technique that creates an object of type `X`, while maintaining the behavior of `XImpl`.

```
1  class X : public XImpl, private XProf {
2  public:
3     X(const XImpl & x) : XImpl(x) { }
4     XBase * Clone() const {
5        XProf::Profile();
6        XBase * x = XImpl::Clone();
7        XBase * newX =
8           new X(*dynamic_cast<XImpl*>(x));
9        delete x;
10       return newX;
11    }
12 };
```

Line 3 in the figure above states that the class `X` can be created by copying its base class, `XImpl`, and nothing else. Line 6 executes the code for this method in the base class, and returns a pointer to an object whose dynamic type is `XImpl`. Lines 7 & 8 create a new object of type `X`, which is a copy of the object created from the previous statement. The pointer must be cast to the same type as the object before it is dereferenced to ensure that the correct constructor is called.

The second problem results from function overloading. Consider a class `XImpl`, with a method `f()` that is overloaded to accept either an `int` or a `string` parameter. `X` must provide implementations of `f()` for both `int` and `string`[2]. If only one implementation for `f()` is provided in `X`, then the other implementation is hidden in the derived class.

8 Conclusions and Future Work

We have described our use of the Decorator pattern to profile object-oriented applications. The pattern is formalized as a correctness preserving transformation to permit non-intrusive profiling, without perturbing the original program. Our case study compares invariant validation at various points in a program. The incremental nature of our case study application causes the invariants to be invalid initially, but as execution of the application progresses the invariants eventually become valid. This progression from invalid to valid has prompted us to investigate the notion of temporal invariants that are *always*, *eventually* or *never* true [13].

Our ongoing work includes extending the invariant validation to other types of applications where objects in the program are more volatile. Moreover, we are constructing a case study to apply the Decorator pattern to profiling memory in C++ applications.

[2] See name hiding in reference [30], or reference [18].

Proceedings of the 11 th IEEE International Workshop on Program Comprehension (IWPC'03)
1092-8138/03 $17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] S. Alexander. The C++ resources network. http://www.cplusplus.com, October 2001.

[3] Z. Aral and I. Gertner. Non-intrusive and interactive profiling in parasight. *Proceedings of the ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems*, 23(9):21–30, 1988.

[4] AT&T Staff. *The Unix (System V) Programmer's Manual*. Prentice-Hall, 1990.

[5] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[6] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57, Paris, France, December 1996.

[7] Boldsoft, Rational Software Corp, IONA and Adaptive Ltd. Response to the UML 2.0 OCL RfP. Technical report, OMG Document ad/2002, March 1 2002.

[8] F. J. Cavalier. Debugging PHP using a C++ compiler. *Dr. Dobbs Journal*, pages 42–46, March 2002.

[9] E. Duesterwald and V. Bala. Software profiling for hot path prediction: less is more. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 202–211, Cambridge, MA, November 2000.

[10] E. B. Duffy, J. P. Gibson, and B. A. Malloy. Applying the decorator pattern for non-intrusive profiling of object-oriented software. Technical Report CU-7, Clemson University, September 2002.

[11] S. H. Edwards. Making the case for assertion checking wrappers. *RESOLVE Workshop 2002*, pages 28–42, June 2002.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[13] T. H. Gibbs and B. A. Malloy. Weaving aspects into C++ applications for validation of temporal invariants. In *Proceedings of 7th European Conference on Software Maintenance and Reengineering*, Benevento, Italy, March 2003. (to appear).

[14] T. H. Gibbs, B. A. Malloy, and J. F. Power. Automated Validation of Class Invariants In C++ Applications. *Proceedings of the International Conference on Automated Software Engineering (ASE'2002)*, pages 205–214, September 2002.

[15] J. P. Gibson. Formal object oriented development of software systems using LOTOS. Technical Report CSM-114, Stirling University, August 1993.

[16] J. P. Gibson, T. F. Dowling, and B. A. Malloy. The Application of Correctness Preserving Transformations to Software Maintenance. *Proceedings of the International Conference on Software Maintenance (ICSM'2000)*, pages 108–119, October 2000.

[17] P. Gibson and D. Mery. Always and eventually in object models. *In ROOM2*, June 1998.

[18] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[19] R. J. Hall. Call path refinement profiles. *IEEE Transactions on Software Engineering*, 21(21):481–496, June 1995.

[20] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Journal of Software Testing, Verification, and Reliability*, 10(3), September 2000.

[21] ISO/IEC JTC 1. *International Standard: Programming Languages - C++*. Number 14882:1998(E) in ASC X3. ANSI, first edition, September 1998.

[22] O. Kiselyov. Fast Fourier transform. *Free C/C++ Sources for Numerical Computation*, March 2002. http://cliodhna.cop.uop.edu/~hetrick/c-sources.html.

[23] D. E. Knuth. An empirical study of Fortran programs. *Software, Practice & Experience*, 1:105–133, 1971.

[24] B. A. Malloy, T. H. Gibbs, and J. F. Power. Decorating tokens to facilitate recognition of ambiguous language constructs. *Software, Practice & Experience*, 2002. (to appear).

[25] R. Martin. Acyclic visitor. Technical Report wucs-97-07, Washington University Technical Report, September 4–6 1996.

[26] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, second edition, 1997.

[27] J. F. Power and B. A. Malloy. Symbol table construction and name lookup in iso C++. In *Technology of Object-Oriented Languages and Systems, TOOLS 2000*, pages 57–68, Sydney, Australia, November 2001.

[28] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. *ACM Software Engineering Notes*, 22(6):432–439, November 1997.

[29] V. Shoup. Number theory library. http://www.shoup.net/ntl/, March 2002.

[30] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[31] SunPro. Performance tuning an application. *Sun Microsystems*, 1992.

[32] J. M. Vlissides and M. A. Linton. IV tools. http://www.vectaport.com/ivtools/, March 2002.

[33] B. Wampler. The V C++ GUI framework. http://www.objectcentral.com, October 2001.

Proceedings of the 11 th IEEE International Workshop on Program Comprehension (IWPC'03)
1092-8138/03 $17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY